PACE: Proactively-Secure Accumulo with Cryptographic Enforcement

Scott Ruoti, Ariel Hamlin, Emily Shen, Cassandra Sparks, Robert Cunningham
MIT Lincoln Laboratory
Lexington, Massachusetts
scott.ruoti@ll.mit.edu, *ariel.hamlin@ll.mit.edu*, *emily.shen@ll.mit.edu*, *cass.a.sparks@gmail.com*, *rkc@ll.mit.edu*,

***Abstract*—Cloud-hosted databases have many compelling benefits, including high availability, flexible resource allocation, and resiliency to attack, but it requires that cloud tenants cede control of their data to the cloud provider. In this paper, we describe Proactively-secure Accumulo with Cryptographic Enforcement (PACE), a client-side library that cryptographically protects a tenant's data, returning control of that data to the tenant. PACE is a drop-in replacement for Accumulo's APIs and works with Accumulo's row-level security model. We evaluate the performance of PACE, discussing the impact of encryption and signatures on operation throughput.**

I. Introduction

Over the last several years, many companies have moved their infrastructure to the cloud. This move is motivated by the cloud's increased availability, flexibility, and resilience [1]. Most importantly, the cloud enables a level of availability and performance that would be impossible for many companies to achieve using their own infrastructure. For example, using a cloud infrastructure Kepner et al. achieved over 100,000,000 database inserts per second [2].

While the benefits of the cloud are compelling, they are offset by the lack of control. No longer is a company the sole administrator of its data and services, but rather it relies on IT support from the cloud provider. While cloud providers have strict policies preventing cloud administrators from accessing or modifying tenants' data, tenants are unable to ensure that these policies are followed. While relying on the cloud provider's promised security is sufficient for some use cases, it is insufficient for sensitive data—for example, personally-identifying information or intellectual property.

In this work, we describe how cryptography can be used to enable organizations to store their data on the cloud while retaining control of that data. To demonstrate the feasibility of this approach, we build a client-side library that encrypts and signs data prior to uploading the data to Accumulo cloud servers: Proactively-secure Accumulo with Cryptographic Enforcement (PACE). This library allows cloud tenants to control the visibility of their data using encryption and ensure that other parties cannot insert or modify data using digital signatures.

DISTRIBUTION STATEMENT A. Approved for public release: distribution unlimited.

This material is based upon work supported by the Department of Defense under Air Force Contract No. FA8721-05-C-0002 and/or FA8702-15-D-0001. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Department of Defense.

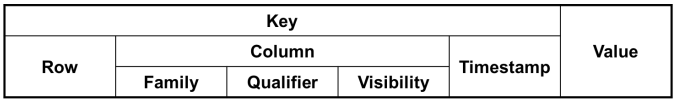


Fig. 1: Accumulo cell schema

The contributions of our work include:

- **Cryptographic enforcement of access control.** The PACE library allows organizations to encrypt all data stored in an Accumulo table using a single key. Alternatively, the PACE library supports cryptographically enforced attribute-based access control (CEABAC), encrypting each row of a table based on the access control policy specified in the row's visibility field.
- textbfRow-level integrity PACE supports the ability to sign data inserted into a table and notifies users when data has been inserted or modified that is not properly signed. Importantly, the PACE library is **not** a research prototype, but a fully fleshed-out implementation ready for use as a drop-in replacement to existing Accumulo APIs.
- **Performance profiling.** We evaluate the PACE library's impact on operational throughput (i.e., inserts/reads per second). This evaluation demonstrates that while encryption and signatures have an impact on throughput, the impact is small enough to be acceptable in many use cases.

II. Background and Related Work

In this section, we describe the Accumulo database software. We also discuss our threat model. Finally, we detail related work.

A. Accumulo

Accumulo is a NoSQL database based on Google's BigTable design [3]. It is primarily concerned with allowing for a high rate of ingest (i.e., inserts).

Tables in Accumulo are an ordered collection of *cells* (see Figure 1). The cell is divided into two sections, the key and the value. The key is composed of a row, a column family,

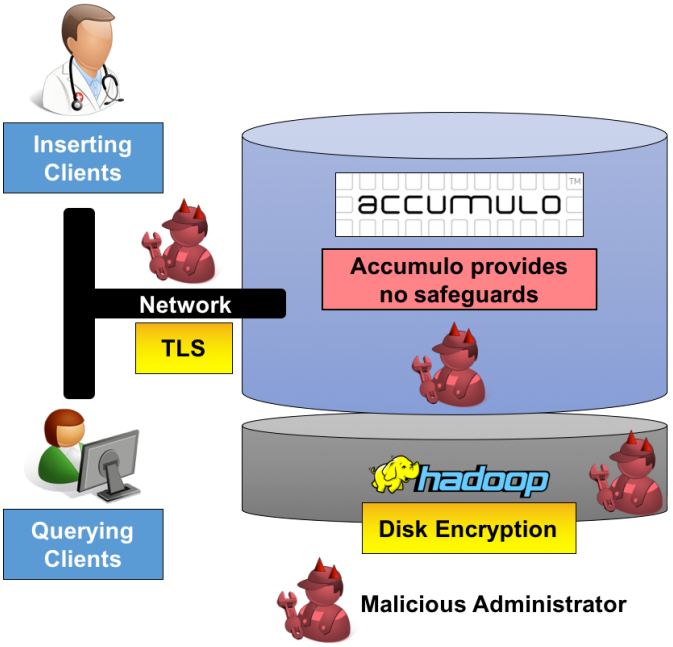


Fig. 2: Threat Model

a column qualifier, and a visibility field. The row, column family, and column qualifier are used to to order and group cells in Accumulo. The way in which these three fields are used determine the "schema" of the table, though this schema is not a strict SQL-like schema.

The visibility field is used to provide fine-grained access control to individual cells. It stores a binary expression over attributes—for example, `doctor OR (nurse AND admin)`. When Accumulo is queried, it only returns cells for which the querying user has been assigned sufficient attributes to satisfy the visibility field's binary expression.

The timestamp orders cells that have identical row, column family, and column qualifier values. By default, when cells share row, column family, and column qualifier values, only the cell with the most recent timestamp is returned by the Accumulo server. Finally, the value is used to store arbitrary binary data.

B. Threat Model

The cloud database threat model has five entities—inserting clients, querying clients, the network, the database software (e.g., Accumulo), and the disk storage platform (e.g., Hadoop). The security of the cloud database can be compromised at any of these entities. While the cloud tenant can control the security of their own clients and ensure that TLS is used, the tenant has little to no control of the administration of either the database software or the disk storage platform.

In this paper we focus on addressing the threat of a malicious database software administrator. While the techniques discussed in this paper also provide some protection against a compromised network or malicious disk storage administrator, this is tangential to our goals. Finally, we consider malicious clients — either inserting or querying —to be out of scope.

C. Related Work

Encrypted databases have received significant attention in the research community in recent years. Advanced cryptographic techniques such as fully homomorphic encryption (FHE) [4] can provide a solution, but they are still prohibitively impractical for real use cases. CryptDB [5], a system for encrypting SQL databases in a way that still allows for queries to be executed, is perhaps the most well known technology that leverages simpler cryptographic techniques to provide protection. Arx [6] improves on this design by using stronger encryption (achieving IND-CPA, or indistinguishability under chosen plaintext attack) and offering more functionality. A systematization of research into cryptographically protected database search was conducted by Fuller et. al [7].

These efforts towards enabling querying over cryptographically protected data are related to but separate from the problem of enforcing record-level access controls over databases. The former aims solely to protect against malicious database administrators while the latter also aims to prevent individual records in the database from being accessed by unauthorized queriers. Attribute-based encryption (ABE) has been extended to enforce access control policies [8]; the use of simpler cryptographic techniques to achieve similar goals has been studied by Solomon et. al [9].

III. Architecture

Proactively-secure Accumulo with Cryptographic Enforcement (PACE) is a Java library that cryptographically protects data on the client. This prevents Accumulo cloud administrators from viewing, inserting, or modifying sensitive data. PACE is a drop-in replacement for the existing Accumulo API, only requiring developers to replace a single line of code—the instantiation of `Writer` and `Scanner` classes—and provide appropriate configuration files and the user's set of encryption and signature keys. The problems of key management and distribution are not handled by the library. PACE can be downloaded at https://github.com/mit-ll/PACE.

A. Encryption

To protect an administrator from viewing sensitive data, the PACE library encrypts all data. The row, column family, column qualifier, and value field are encrypted while the visibility and timestamp fields are not encrypted as they are required to ensure that Accumulo functions as intended. Each of the four fields are processed independently, allowing only a subset to be encrypted or specifying different encryption options for each field. PACE supports three types of encryption—field-level encryption, searchable encryption, and visibility-based encryption.

1) Field-level Encryption: Field-level encryption is the most basic type of encryption, and searchable and visibility-based encryption both build off of it. In field-level encryption, one or more source fields (row, column family, column qualifier,

and value) are concatenated together, encrypted, then stored in the destination field. The content of the source fields is then removed. During decryption, this process is reversed, and the source fields will be replaced with the values from the decrypted destination field.

PACE encrypts data using AES and supports the following modes: CTR, CFB, CBC, OFB, and GCM. For each of these modes, all appropriate key sizes are supported—128, 192, 256. The key used to encrypt data is selected using a `KeyId`, allowing different fields to be encrypted with different keys.

Because each of the supported modes uses a random initialization vector (IV) or nonce, the same data will encrypt to different ciphertext. This prevents inference-based attacks [10], but also prevents encrypted fields from being searchable.

2) Searchable Encryption: PACE also support searching for encrypted data. This is done using AES in SIV mode [11] to provide deterministic encryption—i.e., the same plaintext will always encrypt to the same ciphertext.

When a specific value is searched for in an encrypted field (e.g., find encrypted row "Alphabet"), then the search term is encrypted deterministically, and that term is searched on the server. Because AES does not preserve the lexicographical ordering of plaintext in the ciphertext, searches over a range of encrypted values (e.g., get all rows between "A" and "E") cannot be encrypted and sent to the server for filtering. Instead, all possibly matching entries are downloaded by the client and filtered after decryption. This process is handled automatically by the API, and the developer does not need to worry where filtering will happen.

Order-preserving encryption would have allowed full server-side searching of encrypted data [12], but order-preserving encryption has other drawbacks (e.g., poor performance and information leakage [13]). Similar to, though less severe than order-preserving encryption, PACE's searchable encryption is susceptible to inference attacks, and is not suitable for data with a known distribution [10].

3) Visibility-Based Encryption: Accumulo controls access to cells based on binary expressions over attributes—for example, `doctor OR (nurse AND admin)`—called the visibility expression. PACE supports encrypting fields not with a global key (field-level and searchable encryption), but based on this visibility expression. We have named this functionality cryptographically enforced, attribute-based access control (CEABAC). CEABAC is implemented using access trees [8]. Using an identity-based encryption scheme [8] would be prohibitively slow, and instead CEABAC is implemented using symmetric encryption. This results in significantly increased performance at the cost of collusion-resistance (which is out of scope based on our threat model). CEABAC is more fine-grained than either field-level or searchable encryption, ensuring that a client not only has the right key for the table, but also keys for the attributes in the column visibility field.

To implement CEABAC, each Accumulo attribute (i.e., authorization) must have an associated attribute key.[1] When

[1]The attribute keys can also be identified using a `KeyId` allowing for multiple keys for a given attribute.

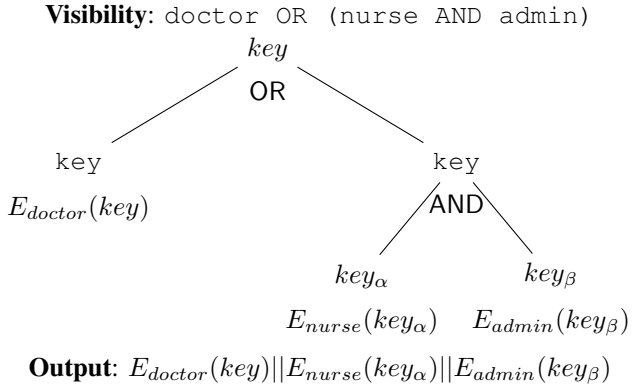


Fig. 3: CEABAC Encryption of a Key

encrypting a cell, a randomly generated key is used to encrypt the cell, and then the randomly generated key is encrypted using the appropriate attributes. `AND` first splits the random key using additive secret sharing using XOR, where each share is then encrypted with one of the attribute keys. `OR` duplicates the key, with each duplicate key being encrypted with a different attribute key. This process occurs recursively to support arbitrary binary expressions, and the final result is a collection of all the shares generated by processing the visibility field. An example encryption is shown in Figure 3.

B. Signatures

To prevent an administrator from surreptitiously inserting or modifying data, it is sufficient to digitally sign the data prior to insertion. Signing data has the added benefit of providing authenticity protection (i.e., attributing an author to each cell in an Accumulo table). While signatures enable detection of the insertion or modification of data, they cannot detect the removal of data by the administrator. To accomplish this, authenticated data structures can be used [14], though this functionality is not currently supported by the PACE library.

To sign a record, the entire Accumulo cell is first hashed using SHA-256. The hash of the cell is then signed using one of the following signature algorithms: RSA with PKSC1 padding, RSA with PSS padding, DSA, or ECDSA. Finally, the signed hash is stored in one of three locations:

1) **Value field.** The signed hash is prepended to the cell's value field. This is the default storage location. Clients using the standard Accumulo API to query the database will see the hashes in their query results and be unable to interpret them.
2) **Separate table.** The signed hash is stored in a separate table. This second table mirrors the first value, except that the value field is replaced with the respective cell's signed hash. This storage scheme allows for clients using the standard Accumulo API instead of PACE to still read data as usual (without being able to verify its integrity or authenticity, of course). This method has storage and query overhead related to creating and querying a second table in lockstep with the first.

3) **Column visibility field.** The signed hash is appended as an `OR` clause to the existing visibility field—i.e., `(visibility) OR SIGNED_HASH`. Tables using this storage scheme can be read by both PACE and non-PACE clients, as non-PACE clients can safely ignore the new visibility attribute. This method is more efficient than storing data in a separate table, but it interferes with Accumulo's cell versioning system and therefore should only be used if versioning is turned off for the table.

C. Key Management

The PACE library does not included a key management system. Instead, PACE takes as input a key container that is responsible for managing a user's keys. The PACE library includes implementations for containers (one for encryption, one for signatures) that store and retrieve a user's keys from the local file system. Also included are scripts for generating and updating these key containers. Developers are free to develop alternative containers—for example, a key container that stores a local cache of the user's keys but also queries a key management service (KMS) when additional keys are needed.

IV. Evaluation

We benchmarked PACE against a cloud-hosted Accumulo instance with a single tablet server. The benchmarks were run on commodity hardware (2014 MacBook Pro) and were executed using the Java Microbenchmark Harness (JMH).[2] We measured the performance of PACE's encryption and signature functionality, both for reading and writing records, and for a various set of parameters.

For each benchmark, performance is measured as an average of 250 runs. First, 5 warmup runs are executed to ensure that the Java JVM was in a stable state. Next, 25 measurements runs were conducted. Finally, this process was repeated (i.e., forked) in 10 separate JVM instances.

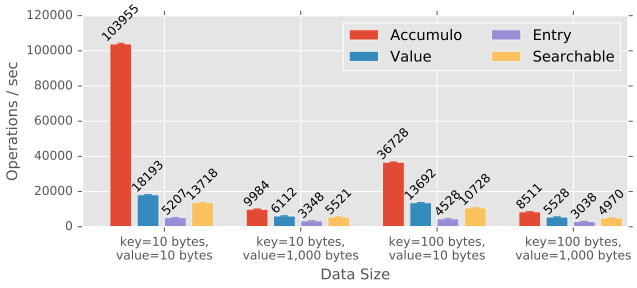
For each fork, the necessary Accumulo tables were created, and after the fork they were deleted. All the data used was randomized, but it was generated using a known seed to ensure that the tests all used the same random data. In each run, we measured the total time taken to complete the run, including the time to setup PACE and to read or write the records. Instead of reporting on total time, we report on throughput—the number of reads or writes per second.

The purpose of this evaluation is to compare the performance of PACE-enabled versus non-PACE-enabled clients. Specifically, it does not explore parallelizing the client software or increasing the number of tablets.

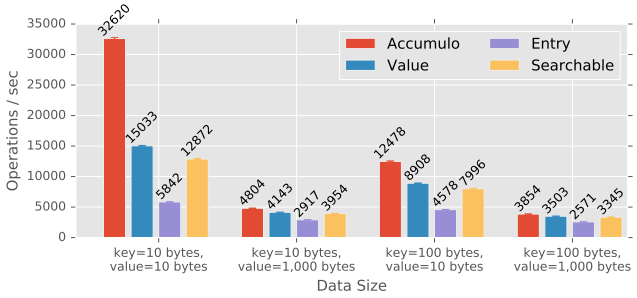
1) Paramaters: We tested the following parameters:

- **Size of the key fields.** We tested making the key fields—row, column family, and column qualifier—small (10 bytes) and large (100 bytes).
- **Size of the value field.** We tested making the value field small (10 bytes) and large (1,000 bytes).

[2]JMH allows for fine-grained measurements at microsecond granularity.



(a) Write



(b) Read

Fig. 4: Encryption performance by configuration and data size

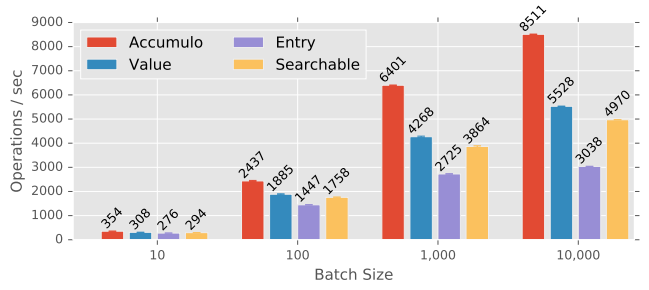
- **Number of rows.** We tested reading/writing various numbers of rows in each operation—1 row, 10 rows, 100 rows, and 1,000 rows.
- **Number of columns**. For each row read/written, we varied the number of columns inserted—1 or 10 columns. The total number of entries written equals number of rows multiplied by number of columns.

2) Encryption: For encryption, we tested the following configurations: Accumulo with no encryption (Accumulo), field-level encryption of a single column (Field-level), encrypting a single column with CEABAC (CEABAC-Column) encrypting the whole entry with CEABAC (CEABAC-Entry), and encrypting the key with searchable encryption and the value with CEABAC (Searchable).

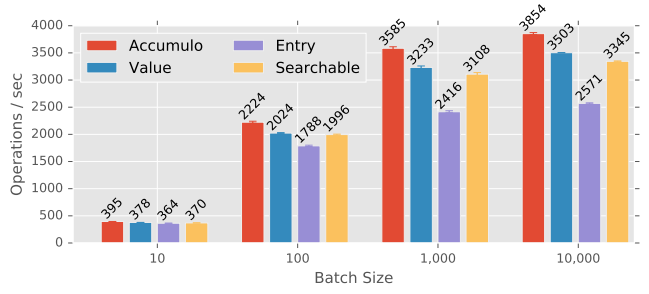
3) Signatures: For signatures, we tested the following configurations: Accumulo with no signatures (Accumulo), storing the signature in the value field (Value), storing the signature in the column visibility field (Visibility), and storing the signature in another table (Table). We also compared various signature algorithms: RSA with PKSC1 padding (RSA-PKSC1), RSA with PSS padding (RSA-PSS), DSA (DSA), and ECDSA (ECDSA).

A. Encryption Results

Figure 4 gives the performance statistics as we vary the size of the data read/written. In this figure, the batch size has been held constant (1000 rows, 10 columns). From this figure, it is clear that encryption with PACE has a significant impact on performance—in the worst case, it reduces the rate of inserts by a factor of 20 and reads by a factor of 6. Still, for writes

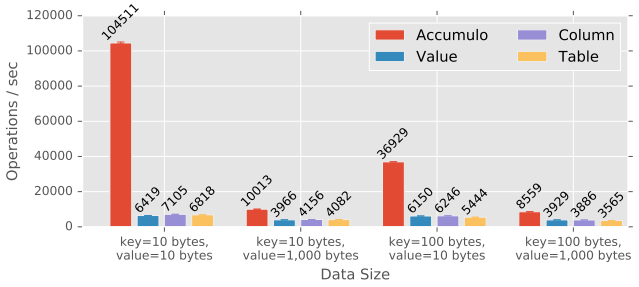


(a) Write

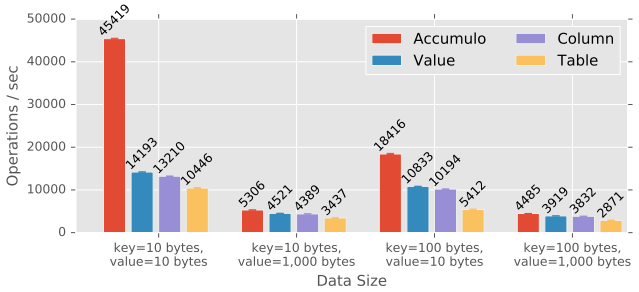


(b) Read

Fig. 5: Encryption performance by configuration and batch size



(a) Write



(b) Read

Fig. 6: Signature performance by configuration and data size

with higher data-size, PACE's performance is reasonable—in the best case, 35% reduction for writes and 10% for reads.

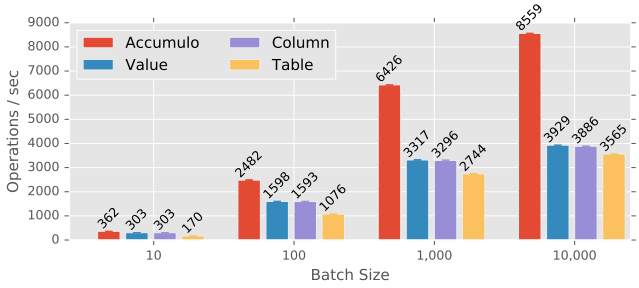
Figure 5 also shows the comparative performance cost of each encryption configuration. In general, the fewer fields encrypted, the faster the encryption. Also, CEABAC is slower than using field-level searchable encryption (compare "Searchable" and "Entry" performance). This is due to the fact that CEABAC requires many possible encryption operations—one per attribute in the visibility expression.

Figure 5 describes what happens as the number of writes/reads is varied (row count multiplied by a constant column count of 10) as the data size is held constant (100 byte key fields, 1000 byte value field). This figure shows that for low numbers of operations, Accumulo's overhead dominates PACE's overhead. Still, as the number of operations increases, PACE's overhead becomes more significant, although the difference is not nearly as drastic as varying size—in the worse case, 65% reduction for writes and 35% for reads.

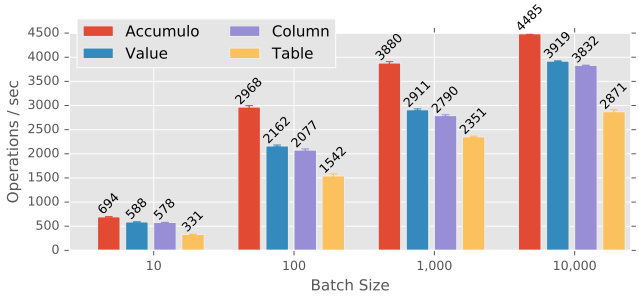
Based on our performance measurements, we note that PACE has reasonable overhead in the following instances. First, when data sizes are large, this consideration dominates all others, as for large data sizes PACE has reasonable overhead even when other parameters are less favorable. Second, when the number of operations is relatively small, PACE's overhead is tolerable. Finally, PACE may be suitable when a high rate of reads is more important than a high rate of writes.

B. Signature Results

Figure 6 and Figure 7 give the performance metrics for each of the different signature configurations. Signatures all have

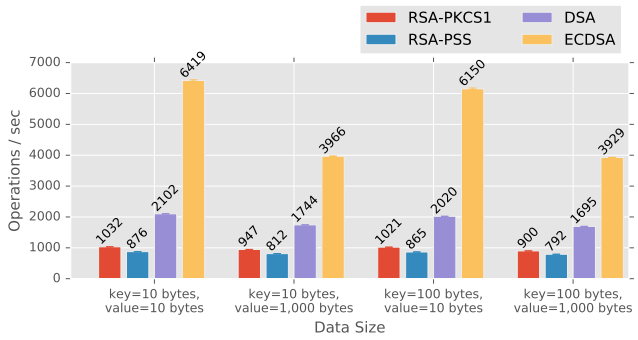


(a) Write

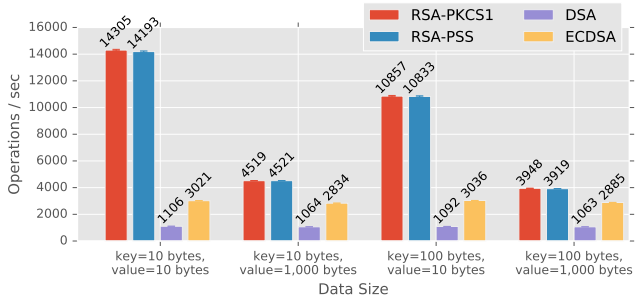


(b) Read

Fig. 7: Signature performance by configuration and batch size

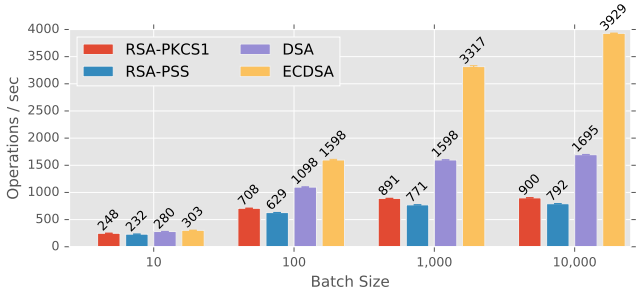


(a) Write

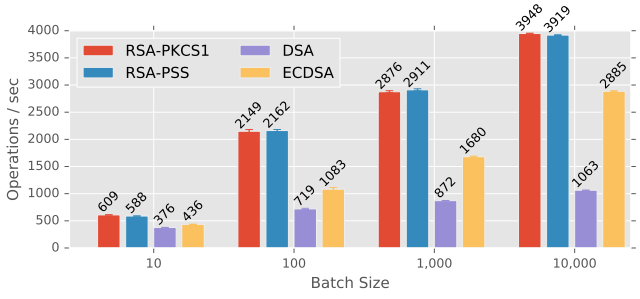


(b) Read

Fig. 8: Signature performance by algorithm and data size



(a) Write



(b) Read

Fig. 9: Signature performance by algorithm and batch size

a significant performance impact, though as usual as the data size or batch size increases, the performance overhead also decreases. Interestingly, the different signature configurations all perform similarly, though there is still a strict ordering on performance ordering— (from best performing to worst) storing signatures in the value, storing signatures in the column visibility, and storing signatures in a separate table.

Figure 8 and Figure 9 compares the various different signature algorithms. For writing entries, ECDSA has by far the highest performance, with RSA having the worst performance. The situation is reversed for reading entries, with ECDSA being slower and RSA being faster; still, as the data size increases, ECDSA's and RSA's performance become more similar. DSA's performance is poor for both reads and writes.

V. Summary

In this paper, we described Proactively-secure Accumulo with Cryptographic Enforcement (PACE), a library for protecting data stored in Accumulo, giving cloud tenants back control of their data. PACE is a fully functioning drop-in replacement for the existing Accumulo API. We measured PACE's performance and found that while it does impact performance, this impact is minimized for entries with large data sizes, low volumes of read and writes, and for read operations in general. While PACE does add significant overhead, it still allows for throughput that would be acceptable for many applications.

Acknowledgment

The authors would like to thank Ben Kaiser, Bryan Richard, Sarah Scheffler, Mayank Varia, and Arkady Yerukhimovich for their earlier work on the PACE project.

References

[1] S. UK, "Why move to the cloud? 10 benefits of cloud computing," https://www.salesforce.com/uk/blog/2015/11/why-move-to-the-cloud-10-benefits-of-cloud-computing.html, accessed: 2017-04-15.

[2] J. Kepner, W. Arcand, D. Bestor, B. Bergeron, C. Byun, V. Gadepally, M. Hubbell, P. Michaleas, J. Mullen, A. Prout *et al.*, "Achieving 100,000,000 database inserts per second using accumulo and d4m," in *High Performance Extreme Computing Conference (HPEC), 2014 IEEE*. IEEE, 2014, pp. 1–6.

[3] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Transactions on Computer Systems (TOCS)*, vol. 26, no. 2, p. 4, 2008.

[4] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *Proceedings of the Forty-first Annual ACM Symposium on Theory of Computing*, ser. STOC '09. New York, NY, USA: ACM, 2009, pp. 169–178. [Online]. Available: http://doi.acm.org/10.1145/1536414.1536440

[5] R. A. Popa, C. Redfield, N. Zeldovich, and H. Balakrishnan, "Cryptdb: protecting confidentiality with encrypted query processing," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM, 2011, pp. 85–100.

[6] R. Poddar, T. Boelter, and R. A. Popa, "Arx: A strongly encrypted database system," Cryptology ePrint Archive, Report 2016/591, 2016, http://eprint.iacr.org/2016/591.

[7] B. Fuller, M. Varia, A. Yerukhimovich, E. Shen, A. Hamlin, V. Gadepally, R. Shay, J. D. Mitchell, and R. K. Cunningham, "Sok: Cryptographically protected database search," https://arxiv.org/pdf/1703.02014.pdf, 2017.

[8] V. Goyal, O. Pandey, A. Sahai, and B. Waters, "Attribute-based encryption for fine-grained access control of encrypted data," in *Proceedings of the 13th ACM conference on Computer and communications security*. Acm, 2006, pp. 89–98.

[9] M. G. Solomon, V. Sunderam, and L. Xiong, *Towards Secure Cloud Database with Fine-Grained Access Control*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 324–338. [Online]. Available: http://dx.doi.org/10.1007/978-3-662-43936-4_21

[10] M. Naveed, S. Kamara, and C. V. Wright, "Inference attacks on property-preserving encrypted databases," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 644–655.

[11] D. Harkins, "Synthetic Initialization Vector (SIV) Authenticated Encryption Using the Advanced Encryption Standard (AES)," RFC 5297, Oct. 2008. [Online]. Available: https://rfc-editor.org/rfc/rfc5297.txt

[12] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu, "Order preserving encryption for numeric data," in *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*. ACM, 2004, pp. 563–574.

[13] A. Boldyreva, N. Chenette, Y. Lee, and A. O'neill, "Order-preserving symmetric encryption," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2009, pp. 224–241.

[14] C. Martel, G. Nuckolls, P. Devanbu, M. Gertz, A. Kwong, and S. G. Stubblebine, "A general model for authenticated data structures," *Algorithmica*, vol. 39, no. 1, pp. 21–41, 2004.